Verifying BDD Algorithms through Monadic Interpretation*

Sava Krstić[1] and John Matthews[2]

[1] Oregon Graduate Institute krstic@cse.ogi.edu
[2] Compaq Cambridge Research Lab John.Matthews@compaq.com

Abstract. Many symbolic model checkers use *Binary Decision Diagrams* (BDDs) to efficiently determine whether two Boolean formulas are semantically equivalent. For realistic problems, the size of the generated BDDs can be enormous, and constructing them can easily become a performance bottleneck. As a result, most state-of-the-art BDD programs are written as highly optimized imperative C programs, increasing the risk of soundness defects in their implementation. This paper describes the use of *monadic interpreters* to formally verify BDD algorithms at a higher level of abstraction than the original C program, but still at a concrete enough level to retain their essential imperative features. Our hope is then that verification of the original C program can be achieved by strictly localized refinement reasoning.
During this work we encountered the surprising fact that modeling imperative recursive algorithms monadically often results in logical functions that are both partial and nestedly-recursive in their (hidden) state parameters, making termination proofs difficult.

1 Introduction

Confidence in results produced by verification tools varies. Counterexamples we can directly check. A theorem prover's claim of the validity of a formula can be checked by an independent tool that tests the validity of derivations recorded in proof scripts. But when a model checker says that a formula is true, such independent checking is untenable for large examples. Since large examples are what model checkers are made for, trusting their results seems tantamount to trusting correctness of their design.

Model checkers are usually built on top of a BDD (binary decision diagrams) package, or some other set of efficiently implemented algorithms for representing and manipulating boolean formulas. Verifying the correctness of a model checker thus naturally splits into two parts: verification of the model checking algorithms assuming correctness of the BDD package, and verification of the BDD package. Recent work of Reif et al. [RRSV00] successfully carried out the first verification

* The research reported in this paper was supported by the National Science Foundation Grants EIA-0072761 and CDA-9703218, Compaq Computer Corporation, and Intel Corporation.

task for the model checker RAVEN. The focus of this work is on the second task and our goal is to provide a technique for proving correctness of high performance BDD packages.

The efficiency of modern BDD programs is achieved at the expense of often highly complex code structure, for example by implementing custom hash tables and garbage collection routines, employing tricks with unused bits in pointers, and so on. To avoid runtime overhead, the code is written in a low level language, usually C. Our goal of formally verifying such algorithms *as originally written* distinguishes our work from other proofs of BDD algorithms [HPPR98,VGPA00,Sum00].

BDD libraries are hierarchical: More complex programs are built on top of a set of atomic primitives using standard programming constructs. In this paper we decompose the verification problem accordingly into two steps: verifying an abstraction of the program with the primitives specified axiomatically, and then a refinement proof that the C implementation of the primitives and programming constructs is faithful to the abstraction. This paper concentrates on the first step, although we hope the reader will be convinced that the axioms governing the primitives can be justified by purely local reasoning over their C implementations, and that our logical characterization of the standard programming constructs (e.g., sequencing of statements and use of local variables) is faithful to the corresponding C semantics.

We adopted an abstraction method called *monadic interpretation* for the first step; it is particularly suitable for higher order logic theorem provers such as *Isabelle/HOL* (in the sequel, *Isabelle*). In particular, local C variables that are statically assigned to only once (the common case) are abstracted to *logical* function parameters, allowing the theorem prover to automatically carry out routine inferences about variable creation, renaming, and substitution.

We begin in Section 2 by giving an informal description of a basic BDD package. Monadic interpreters are briefly described in Section 3. The BDD routines are then monadically interpreted as *Isabelle* functions in Section 4. The most complicated of the library programs (`Apply`) is a recursively defined partial function, which presents a difficulty for *Isabelle*, where all functions are total. How we deal with recursion and model the program `Apply` is explained in Section 5. Then in Section 6 we state the correctness properties of non-atomic programs and comment on their proofs. We comment on what is needed to refine our abstraction to C in Section 7.

We hope the relevance of this paper goes beyond its immediate objective. It presents, by means of an extensive example, a method, based on monadic interpretation and refinement, for proving correctness of imperative programs that use complicated recursion, manipulate complex state, and raise exceptions.

2 Basic BDD Package

A binary decision diagram is a rooted directed acyclic graph in which every node represents a boolean function. Two special nodes represent the two constant

functions. Every other node u has an associated variable x and two child nodes l and h. The boolean function represented by the node u is defined recursively by $f_u =$ if x then f_l else f_h. Bryant [Bry86] originally proved that every function is represented by a unique reduced ordered BDD, where *reduced* means that no two nodes represent the same function, and *ordered* means that variable names are totally ordered and that every node's variable name precedes the variable names of its children.

Following the exposition in [And96], we give now a fairly abstract description of a typical implementation of (reduced, ordered) BDDs and a small package of programs, sufficient to define a tautology checker. It contains some underspecified basic types and atomic functions, and a few more complex but fully specified C programs. In the next two sections we will show how all this naturally translates into *Isabelle*.

We abstract the global state used by the BDD package as two tables: `BDD` and `HASH`. The first table represents the storage pool of BDD nodes, and the second is a hash table that memoizes a reverse mappingu from the contents of a node to its address. Node addresses are represented by the abstract type *node*.

Specifically, each entry of `BDD` associates to a node `u` a unique triple `(i,l,h)`, where `i` is a natural number (the *level* of `u`) and `l` an `h` are nodes (the *low* and *high children* of `u`). The level of a node is the position of the node's variable name in the given variable ordering. Each entry in `HASH` maps a triple `(i,l,h)` to a unique node.

There are two special nodes `TrueNode` and `FalseNode`; the atomic procedure `initializeState` replaces the current state with the initial state whose tables associate the special nodes with the triples `(0,TrueNode,TrueNode)` and `(0,FalseNode,FalseNode)` respectively, and contain no other entries.

The accessor functions `lookupLev`, `lookupLow` and `lookupHigh` take a node `u` as an argument, and return the components `i`, `l` and `h` of the triple the table `BDD` associates with `u`. What, if anything, these functions return in the case when `u` is not in the table `BDD` is left unspecified. Similarly, the function `lookupH` takes a triple `(i,l,h)` as input, and returns the node associated by `HASH` to this triple. Again, we do not know what `lookupH` returns if the input triple is not in the table `HASH`. However, there is another function, `member`, which also takes triples `(i,l,h)` as inputs and returns a boolean value that is `True` if and only if the triple is in the table `HASH`.

The simple function `bool2node` maps `True` and `False` to `TrueNode` and `FalseNode` respectively, while `node2bool` does the opposite, being unspecified for "non-boolean" inputs.

The function `getFreshNode` takes no input and returns a node that is not already in the table `BDD`, raising an exception if there are no free nodes left.

The list of atomic programs is completed with `insertNode` and `insertH`. Both take as input a quadruple `(i,l,h,u)` consisting of a level and three nodes, have no output, but change the state. The effect of `insertNode` is the update of `BDD` by an entry that associates `u` with `(i,l,h)`. The `insertH` similarly updates `HASH`.

The remaining programs are defined by combining the atomic ones by means of standard constructs: sequencing, conditionals, and recursion. Shown below is the program `Mk`, the only one in the package that directly calls the updating functions `insertNode` and `insertH`, thus guaranteeing that all higher-level BDD programs preserve some critical properties of the state, for example that the two tables are inverses. (See Section 5 for the complete state invariant.)

```
node Mk(int i, node l, node h) {
  if (l == h)
    return l;
  else if (member(i,l,h))
    return lookupH(i,l,h);
  else {
    node u = getFreshNode();
    insertNode(i,l,h,u);
    insertH(i,l,h,u);
    return u;
  }
}
```

The program `Apply` takes a binary boolean operation *op* and two nodes, and returns a node representing the boolean function one could otherwise obtain by applying *op* to boolean functions represented by the two input nodes. It is the most complicated program in the package and is defined recursively as follows.

```
node Apply(opFn op, node u, node v) {
  int i = lookupLev(u);
  int j = lookupLev(v);
  if (i == j)
    if (i == 0) return bool2node((*op)(node2bool(u),node2bool(v)));
    else {
      node l1 = lookupLow(u);
      node l2 = lookupLow(v);
      node h1 = lookupHigh(u);
      node h2 = lookupHigh(v);
      return Mk(i,Apply(op,l1,l2),Apply(op,h1,h2));
    }
  else if (i < j) {
    node l = lookupLow(v);
    node h = lookupHigh(v);
    return Mk(j,Apply(op,u,l),Apply(op,u,h));
  }
  else {
    node l = lookupLow(u);
    node h = lookupHigh(u);
    return Mk(i,Apply(op,l,v),Apply(op,h,v));
  }
}
```

The program `Apply` is used by another program `Build` to produce a node representing the boolean function defined by a given boolean expression. `Build` is defined by primitive recursion over the structure of the expression. Finally, we

have the program `TautChecker` which just initializes the state, invokes `Build` with its input expression, and returns a boolean value: `True` if and only if the node returned by `Build` is `TrueNode`.

In the next section we will see how all the above code can be more or less directly translated into *Isabelle*, so instead of giving the C code for `Build` and `TautChecker` here, we refer to their *Isabelle* definitions in Section 4.

3 Monadic Interpretation

Variants and extensions of Floyd-Hoare logic [AO97] are the most commonly used frameworks for verifying imperative programs. The complexity of our programs forces to adopt an alternative, more flexible, even if less investigated and automated approach. We proceed by modeling BDD programs as functions in higher order logic, in the style of monadic interpreters [Mog91,LHJ95]. Generally, a *monadic interpreter* translates source programs of input type A and output type B into functions of type $A \Rightarrow M\ B$ in the target functional language, where the type constructor M is a suitable *monad* that encapsulates the *notion of computation* used by the source language. Different source languages get interpreted by means of different monads. The target language for us will be *Isabelle*, and the source language would be a fragment of C large enough to describe BDD programs. For our purposes, a so-called "state with exceptions" monad is the appropriate choice. With it, a program of input type A and output type B gets interpreted as a function that given an element of A and a state returns a new state together with either an element of B or the memory overflow exception.

Postponing the definition of the state type `St`, the definition of the monad is as follows.

```
datatype 'a except = OutOfMem | Rslt 'a

types 'a M = "St ⇒ St × 'a except"

constdefs return :: "'a ⇒ 'a M"                          ("η")
         "return ≡ λa s. (s, Rslt a)"
          bind :: "['a M, 'a ⇒ 'b M] ⇒ 'b M"          (infixr "▷" 60)
         "bind ≡ λm f s. let s' = fst (m s) in  case snd (m s)
                     of OutOfMem  ⇒ (s', OutOfMem)
                      | Rslt a    ⇒ f a s'"
```

Every instance of a monad has two distinguished operations: *return* (η) and *bind* ($\triangleright$). The η operator is used to represent effect-free computations, such as the evaluation of pure expressions and functions. The notion of "effect" varies by monad; in our case an effect is either a change in the global state or the raising of an out-of-memory exception. The $\triangleright$ operator simultaneously captures the notions of program sequencing and local variable declaration. The expression $m \triangleright f$ has the effect of first performing the computation m; assuming m returns

normally, then f is applied to m's return value, which results in a new computation that is then performed. The $\rhd$ operator also ensures that state changes and exceptions are propagated correctly between m and f.

Proper monads are required to obey three algebraic identities: *unit laws*

$$(\eta\ x) \rhd f = f\ x \qquad\qquad m \rhd (\lambda x.\ \eta\ x) = m$$

and *associativity of* $\rhd$

$$m \rhd (\lambda x.\ p[x] \rhd f) = (m \rhd (\lambda x.\ p[x])) \rhd f$$

Here $p[x]$ indicates that the bound variable x is allowed to occur free in p but not in f, since x's scope is being restricted on the right-hand side. It is straightforward to prove the above laws hold for our *Isabelle* definitions of `return` and `bind`. Once proved, the laws can then be used to simplify complex monadic expressions.

The rules of higher order logic guarantee that the logical bound variables, which represent local program variables, are re-scoped and renamed as necessary to maintain program equivalence. The monadic representation of programs also allows program recursion to be naturally modeled as logical recursion, as we will see in Section 5. In the next section, we show how the BDD package primitives are axiomatized as monadic computations in our state-with-exceptions monad.

4 Modeling the Basic Package

The following *Isabelle* code models the global state of our BDD package.

```
types  Level = nat

typedecl Node

record NodeRecord =  lev :: Level  low :: Node  high :: Node

types  BDD = "Node ⇒ NodeRecord"
      HASH = "NodeRecord ⇒ Node"

typedecl St

consts  bdd        :: "St ⇒ BDD"
        hash       :: "St ⇒ HASH"
        activeNode :: "St ⇒ Node ⇒ bool"
        activeRcrd :: "St ⇒ NodeRecord ⇒ bool"
```

Thus, the type `Node` is left undefined and so is `St`, but we know that we can extract the two tables from the state. The intuition that the table `BDD` is mathematically a partial function of type `Node ⇒ NodeRecord` is represented in *Isabelle* (where all functions are total) by declaring the table to be a total function of that type, and specifying its domain of definition separately by the function `activeNode`. Similar remarks apply to the table `HASH`. The important thing to

notice, however, is that whatever concrete implementation of the BDD package we later come up with, it should be possible to define the functions *bdd*, *hash*, *activeNode* and *activeRcrd*.

TrueNode and *FalseNode* are declared as constants of type *Node*, and since this type is unspecified, we add an axiom saying that these two nodes are distinct. The initial values *initBDD*, *initHASH*, *initActiveNode* and *initActiveRcrd* are straighforward to define, and then *initializeSt* is introduced by an axiom.

```
consts initializeSt :: "unit M"
axioms initializeSt_ax :
  "initializeSt s = (s',x) ⟹ bdd s' = initBDD ∧ hash s' = initHASH
   ∧ activeNode s' = initActiveNode ∧ activeRcrd s' = initActiveRcrd"
```

Other atomic functions are also introduced by axioms[1]. We show three; the remaining ones are similar or simpler. (The notation *f (u := a)* used in *insertNode_ax* is for function update.)

```
consts lookupLev :: "Node ⇒ Level M"
axioms lookupLev_ax : "⟦lookupLev u s = (s',p); activeNode s u⟧
     ⟹ s' = s ∧ p = Rslt (lev (bdd s u))"

consts getFreshNode :: "Node M"
axioms getFreshNode_ax :
  "getFreshNode s = (s', Rslt u) ⟹ ¬(activeNode s u)"

consts insertNode :: "Level × Node × Node × Node ⇒ unit M"
axioms insertNode_ax : "insertNode (i,l,h,u) s = (s',p) ⟹
            p             = Rslt ()
          ∧ bdd s'        = (bdd s) (u := (|lev = i, low = l, high = h|))
          ∧ activeNode s' = (activeNode s) (u := True)
          ∧ hash s'       = hash s
          ∧ activeRcrd s' = activeRcrd s"
```

Non-atomic programs *Mk*, *Build* and *TautChecker* are fully specified as follows and the recursively defined *Apply* is discussed in the next section.

```
constdefs Mk ::  "Level × Node × Node ⇒ Node M"
        "Mk ≡  λ(i,l,h).
               if l=h then η l
               else member (i,l,h) ▷ (λx.
                 if x then lookupH (i,l,h)
                 else getFreshNode ▷ (λu.
                      insertNode (i,l,h,u) ▷ (λp.
                      insertH (i,l,h,u))))"
```

[1] There is no danger of inconsistency with these axioms; after the refinement step (Section 7) they will be theorems. We could have also introduced our underspecified functions in a purely definitional manner by means of the ε-operator.

```
consts  Build :: "Exp ⇒ Node M"
primrec  "Build (Var i) = Mk (i+1,TrueNode,FalseNode)"
    "Build (Const b) = bool2node b"
    "Build (Exp' oper e1 e2) =
                  Build e1 ▷ (λu.
                  Build e2 ▷ (λv.
                  Apply (oper,u,v)))"

constdefs TautChecker :: "Exp ⇒ bool M"
          "TautChecker ≡ λe.
             initializeSt ▷ (λx.
             Build e ▷ (λu.
             η (u = TrueNode)))"
```

The input tupe of `Build` is that of boolean expessions:

```
datatype Exp = Var nat | Const bool | Exp' Op  Exp  Exp
```

Note that the variable `Var i` is represented by the level `i+1`; the level zero of the BDD table is reserved for `TrueNode` and `FalseNode`.

5 Modeling Recursive Programs (`Apply`)

The definition of `Apply` and the proof of the corresponding recursion theorem is the most difficult part of this work. Even though *Isabelle* has a sophisticated *recdef* mechanism [NP] for recursive definitions with user-supplied well-founded relation or a measure function to justify termination, this method is difficult to apply in our case, mostly because of nested recursion we have to deal with.

Pondering the definition in Section 2, one realizes that even a hand proof of termination of `Apply` requires effort. The ultimate reason for termination is clear: in an ordered BDD (and only those we would like to consider), the level goes down when passing children nodes, so in all recursive calls of `Apply` the level decreases either for both node arguments, or decreases for the "higher", while the other stays the same. Thus, in order to prove that the arguments decrease in recursive calls, it is necessary to work with a restricted set of states, described by a predicate `goodSt` that needs to be preserved by `Apply`. A workable invariant `goodSt` is the conjunction of three properties: (1) being ordered; (2) having inverse `bdd` and `hash` tables; (3) the two tables associate `TrueNode` and `FalseNode` with `TrueRcrd` and `FalseRcrd` respectively, and among active records only these two have level zero.

In *Isabelle* notation, the type of `Apply` is `Op × Node × Node ⇒ Node M`, which is the same as `Op × Node × Node ⇒ St ⇒ St × Node except`; later we will use the shorthand `Z` for it. The `Op` argument is of little significance here, so `Apply` is practically a function of two arguments of type `Node` and one of type `St`. We can expect `Apply` to terminate only if its `St` argument is good. But there must be restrictions on the node arguments as well: they must be present in the state. We capture these restrictions using the node-state relation `u in s` defined as the

conjunction of *goodSt s* and *activeNode u s*. Thus, *Apply* must be modeled as a partial function, with restrictions on its state and node arguments. The expected recursion theorem takes these restrictions as assumption.

theorem *Apply_Recursion:*
"⟦u in s; v in s⟧ ⟹ Apply (oper,u,v) s = F Apply (oper,u,v) s"

Here *F* is the *Isabelle* function of type *Z ⇒ Z*, obtained by a direct monadic translation of the C code for `Apply` in Section 2. In principle, the *recdef* package can handle recursive definitions of partial functions by defining the partial function as a total function whose value is arbitrary outside its "real" domain of definition, and by proving the recursion theorem with the proviso that the argument belongs to that domain, just as in our example. However, there is an additional difficulty that *recdef* cannot easily deal with, viz. the presence of *nested recursion*—a recursive call whose argument contains another recursive call.

Back to the informal definition of `Apply`, consider for example the second of the three lines in which recursion occurs; in expanded form this piece of code could read like this:

```
node ll = Apply(op,u,l);
node hh = Apply(op,u,h);
node w  = Mk(j,ll,hh)
```

After the first call to `Apply`, the state changes, so perhaps `u` and `h` are not even active indices in the new state's BDD table. How do we know then that the second call terminates? We may say that the new state, being the result of an application of `Apply`, must contain the old state intact. But this is circular reasoning, assuming a property of `Apply` before this function has been defined. Our inductive proof of termination of `Apply` must thus be organized as a simultaneous proof of termination and the desired property that `Apply` increases the state. A careful analysis shows that a stronger invariant *goodFn* (predicate on *Z*) is necessary; it is the conjuction of three properties: (1) increasing state; (2) output node is active in the new state; (3) the level of the output node is not larger than the levels of the input nodes.

Note that nesting is not immediately seen in the definition given in Section 2 because the state is not implicitly mentioned in the program text, being an extra hidden argument. If we made it explicit, the three lines above would look like this:

```
(ll,s1) = Apply'(op,u,l,s)
(hh,s2) = Apply'(op,u,h,s1)
(w,s3)  = Mk'(j,ll,hh,s2)
```

exposing the nesting first in

```
hh = fst(Apply'(op,u,h, snd(Apply'(op,u,l,s))))
```

Generalizing this observation, note that nested recursion occurs in every recursively defined imperative program in which there is a sequence of commands containing two recursive calls.

Nested recursion is difficult to treat by automatic tools. In [Sli00], Slind demonstrates that *recdef* can be used even in such cases, but some additional more or less ad hoc arguing is inevitable to make it work. We devised and formulated in *Isabelle* a systematic approach that reduces the problem of justifying a nested recursive definition and proving the appropriate recursion theorem to two specific proof obligations. To make it work, in addition to the measure function (or well-founded ordering) as in *recdef*, the user has to supply a *specification*—property of the function being defined that is needed to prove termination. A brief explanation of the basic form of the method follows; full development will appear elsewhere [KM].

Given a functional $F : (A \Rightarrow B) \Rightarrow (A \Rightarrow B)$ and a well-founded relation $\prec$ on A, one can prove that F has a unique fixed point (that is, a function $f : A \Rightarrow B$ satisfying $f\ x = F\ f\ x$) if it satisfies the *contraction condition*

$$\forall f\ g\ x.\ (\forall y.\ y \prec x \longrightarrow f\ y = g\ y) \longrightarrow F\ f\ x = F\ g\ x$$

This is a fixed point theorem à la Banach, and we can immediately generalize it by considering a non-empty predicate $S : (A \Rightarrow B) \Rightarrow \mathsf{bool}$ and asserting that if S is invariant under F (that is, $S\ f \longrightarrow S\ (F\ f)$), then a weakened contraction condition

$$\forall f\ g\ x.\ (\forall y.\ y \prec x \longrightarrow f\ y = g\ y) \wedge S\ f \wedge S\ g \longrightarrow F\ f\ x = F\ g\ x$$

guarantees the existence of a fixpoint of F and its uniqueness among functions satisfying S.

The weakened form of the last uniqueness result is a drawback, especially in the application to program semantics. When a recursive program is interpreted as a fixpoint of a functional in HOL, the situation is clean when that fixpoint is unique, and not quite so when we have uniqueness only among the set of functions satisfying a certain predicate. We obtained a satisfying solution by strengthening both the invariance and contraction conditions. It works for invariants specified as input-output relations. Precisely, given F and $\prec$ as before, and given a relation $R : A \Rightarrow B \Rightarrow \mathsf{bool}$, the two conditions

$$\forall f.\ (\forall y.\ y \prec x \longrightarrow R\ y\ (f\ y)) \longrightarrow R\ x\ (f\ x)$$
$$\forall f\ g\ x.\ (\forall y.\ y \prec x \longrightarrow f\ y = g\ y \wedge R\ y\ (f\ y)) \longrightarrow F\ f\ x = F\ g\ x$$

are sufficient to guarantee the existence and (unrestricted) uniqueness of a fixpoint of F.

Finally, we need to generalize the last result to cover the possibility of nontermination of the limit function outside a specified set of inputs. That set being described by a predicate $D : A \Rightarrow \mathsf{bool}$, the final form of the invariance and contraction conditions reads as follows.

$$\forall f.\ (\forall y.\ D\ y \wedge y \prec x \longrightarrow R\ y\ (f\ y)) \longrightarrow R\ x\ (f\ x)$$
$$\forall f\ g\ x.\ D\ x \wedge (\forall y.\ D\ y \wedge y \prec x \longrightarrow f\ y = g\ y \wedge R\ y\ (f\ y)) \longrightarrow F\ f\ x = F\ g\ x$$

Now we can formulate a fixpoint theorem that can be used to justify nested recursive definitions.

Theorem 1. *Suppose the last two conditions are satisfied. Then there exists a function* $f : A \Rightarrow B$ *such that*

$$\forall x.\ D\ x \longrightarrow f\ x = F\ f\ x$$

Moreover, f *is unique in the sense that every other function* g *satisfying this same conditional fixpoint equation satisfies also* $\forall x.\ D\ x \longrightarrow g\ x = f\ x$.

The theorem can be proved in *Isabelle* and then instantiated by taking: (1) F to be the functional defining `Apply`; (2) the well-ordering $\prec$ induced by the measure function given by the maximum of the levels of input nodes in the input state; (3) R to be the input-output relation defining the invariant `goodFn`; (4) the predicate D saying that both input nodes are present in the input state. As a result, we obtain the definition of `Apply` together with the quoted theorem `Apply_Recursion`.

6 Correctness

Once the definition and recursion theorem for `Apply` become available, the proof of its correctness and then the correctness of `Build` and `TautChecker` are straightforward. The only interesting auxiliary function is the interpretation function `IntNode` that specifies how a node and a state containing the node determine a boolean function.

```
consts IntNode :: "Node × St ⇒ BoolFn"
recdef IntNode "measure (λ(u,s). lev' u s)"
  "IntNode us = (let u = fst us; s = snd us in
     if u in s then
       (if u = TrueNode then TrueFn
        else if u = FalseNode then FalseFn
        else λ env. if env ((lev' u s) - 1)
                    then IntNode (low' u s, s) env
                    else IntNode (high' u s, s) env)
    else arbitrary)"
```

As for the new notation, `lev' u s` , `low' u s` and `high' u s` are abbreviations for the components of the record `bdd s u`, the type `BoolFn` is just `"Env ⇒ bool"`, and the environment type `Env` is `"Var ⇒ bool"`.

Correctness of `Apply` asserts the relationship between the interpretations of the two input nodes and the output node. To state it we need the obvious version of `Apply` for boolean functions: `BoolFnApply oper f g env = oper (f env) (g env)`.

```
theorem Apply_Correct: "⟦u in s; v in s; Apply (oper,u,v) s = (s',Rslt w)⟧
   ⟹ IntNode (w,s') = BoolFnApply oper (IntNode (u,s)) (IntNode (v,s))"
```

Correctness of `Build` is the statement saying that the interpretation of the node constructed by `Build` is the boolean function represented (via the obvious function `IntExp`) by the expression given as input to `Build`.

theorem *Build_Correct*: "⟦*goodSt s*; *Build e s* = (*s'*,*Rslt u*)⟧ ⟹
IntNode (*u*,*s'*) = *IntExp e*"

Correctness of the tautology checker is its soundness property:

theorem *TautChecker_Correct*:
"*out* (*TautChecker e*) *s* = *Rslt True* ⟹ *IntExp e* = *TrueFn*"

One can also prove the completeness of the tautology checker, saying that if it terminates with the result *False*, then the input expression is not a tautology.

7 Completing the Refinement

Formalization of the complete ANSI C language is a formidable challenge but within reach, as demonstrated by current work of Norish [Nor98] and Papaspyrou [Pap01]. Ideally, verification of C programs would use such a formalization, but for proving properties of a small set of programs partial formalizations could also be acceptable. BDD programs, for example, can be written in a small fragment of C that we can with little pain interpret monadically in *Isabelle*.

We have already produced translations of non-atomic BDD programs and made them a part of an *Isabelle* theory presented in Section 4. It remains to add translations of atomic programs and derive the correctness of the whole translated package. Since *Isabelle* does not directly support refinement of its theories, we would have to manually modify the theory file described in Section 4 as follows.

First, the unspecified types *St* and *Node* are declared to be equal to the state and node type used by the C functions. The functions *bdd*, *hash*, *activeNode*, *activeRcrd* that extract the abstract tables from the concrete representation of the state should at this point have straightforward definitions. Finally, atomic functions are given their concrete definitions and the axiom we previously had for each of them is now a theorem that needs to be proved.

Clearly, the abstract BDD package of Section 4 can be refined this way to more than one C implementation. The complexity of the implementation will not affect the proof of the top-level correctness; it will only show up in the level of difficulty for the refinement proofs of atomic functions.

Leaving further development to future work, a couple of remarks are in order about features that are critical for good performance, but left out in this work. First is the memoization of *Apply* results. Adding it would indeed complicate some of our proofs, but the main reason for this omission is ongoing work where we expect to formalize a general result about equivalence of recursive programs with their versions optimized by memoization. Then we have garbage collection, omitted in the initial phase of this research for reasons of simplicity and irrelevance for the proof of correctness of the tautology checker. Adding it is possible at almost no cost in changing existing proofs. Concretely, reference counts would be added to node records, together with the pertinent atomic functions. The garbage collector would be called by a refined *getFreshNode*, so we would need to reprove correctness of that routine assuming the correctness of the garbage collector.

8 Related Work

Filliâtre seems to be the first to explore monadic interpretation for verifying imperative programs. In [Fil01], he presents a far-reaching generalization of the Floyd-Hoare method, applicable to programs written in an *ML*-like functional language, with recursion and references. In the framework of the *Coq* theorem prover, he uses a generalized monadic translation and suitable program annotations to automatically generate simpler proof obligations from a given correctness statement. The elegance and power of this system notwithstanding, it is not clear how well it handles programs with nested recursion. It would be interesting to see whether the specification predicates (mentioned in Section 5) we found necessary to prove termination of such functions could be added as annotations in this framework.

Verification of BDD algorithms has been a subject of active research and the papers [HPPR98], [VGPA00] and [Sum00] offer proofs done with proof assitants *PVS*, *Coq* and *ACL2* respectively. Some go beyond our work so far in that they cover memoization and/or garbage collection. A common goal of these papers is to extend the prover with a certified BDD package by means of reflection available in their respective systems. The resulting packages have high-confidence and encouraging performance, though still substantially below those of the corresponding C-coded implementations.

BDD algorithms are modeled in [VGPA00,Sum00] as functional programs in "state-threading" style, while the "perfect hashing" trick used in [HPPR98] makes the state constant. Complexity of the proof effort and the proof assistants' idiosyncrasies imposed limitations on the form of some of these programs could be expressed. For example, extra counter parameters are used in [VGPA00] for recursive BDD programs, even though they are not needed for the algorithm being defined. In contrast, our work is an attempt to verify BDD algorithms "in the wild", that is, expressed as closely as possible to the form they appear in existing C implementations. Thus, we have emphasized monadic style and worked hard to allow natural program definitions even if they involve nested recursion.

9 Conclusions

We have made progress towards our ultimate goal of verifying current C language-based model checkers. Adopting the monadic interpretation technique, we have defined an abstract version in *Isabelle* of the imperative code implementing standard BDD algorithms. At this level, we proved correctness of the BDD programs, including the BDD tautology checker. A shallow embedding of a small fragment of C will allow interpretation of the actual C code. By refinement, verifying correctness of the C code will be, in virtue of theorems presented in this paper, reduced to proving precisely identified properties of (only) the atomic BDD functions in the C package.

Acknowledgments

We thank John Launchbury and anonymous referees for useful comments on the paper.

References

[And96] H. R. Andersen. An Introduction to Binary Decision Diagrams. Internet, September 1996.

[AO97] K. R. Apt and E.-R. Olderog. *Verification of sequential and concurrent programs*. Springer-Verlag, 1997.

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Fil01] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 2001.

[HPPR98] F. W. von Henke, S. Pfab, H. Pfeifer, and H. Rueß. Case Studies in Meta-Level Theorem Proving. In J. Grundy and M. Newey, editors, *Proc. Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLS)*, Lecture Notes in Computer Science, pages 461–478. Springer LNCS 1479, September 1998.

[KM] S. Krstić and J. Matthews. Nested recursive definitions in *Isabelle/HOL*. In preparation.

[LHJ95] S. Liang, P. Hudak, and M. P. Jones. Monad transformers and modular interpreters. In *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 333–343, New York, NY, USA, January 1995. ACM Press.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[Nor98] M. Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge Computer Laboratory, 1998.

[NP] T. Nipkow and L. Paulson. *Isabelle/HOL* tutorial.

[Pap01] N. S. Papaspyrou. Denotational semantics of ANSI C. *Computer Standards and Interfaces*, 23:169–185, 2001.

[RRSV00] W. Reif, J. Ruf, G. Schellhorn, and T. Vollmer. Do you trust your model checker? In W. A. Hunt Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design (FMCAD)*. Springer LNCS 1954, November 2000.

[Sli00] K. Slind. Another look at nested recursion. In M. Aagaard and J. Harrison, editors, *Proc. Intl. Conf. on Theorem Proving in Higher Order Logics (TPHOLS)*, Lecture Notes in Computer Science, pages 498–518. Springer LNCS 1869, August 2000.

[Sum00] R. Sumners. Correctness proof of a BDd manager in the context of satisfiability checking. Technical Report TR-00-29, The University of Texas at Austin, Department of Computer Sciences, November 2000.

[VGPA00] K. N. Verma, J. Goubalt-Larrecq, S. Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In J. He and M. Sato, editors, *Proc. 6th Asian Computing Science Conference (ASIAN)*, Lecture Notes in Computer Science, pages 162–181. Springer LNCS 1961, November 2000.